Automatic Generation of Self-monitoring MASs from Multiparty Global Session Types in Jason

Davide Ancona[1], Sophia Drossopoulou[2], and Viviana Mascardi[1]

[1] DIBRIS, University of Genova, Italy
{davide.ancona,viviana.mascardi}@unige.it
[2] Imperial College, London, UK
scd@doc.ic.ac.uk

Abstract. Global session types are behavioral types designed for specifying in a compact way multiparty interactions between distributed components, and verifying their correctness. We take advantage of the fact that global session types can be naturally represented as cyclic Prolog terms - which are directly supported by the Jason implementation of AgentSpeak - to allow simple automatic generation of self-monitoring MASs: given a global session type specifying an interaction protocol, and the implementation of a MAS where agents are expected to be compliant with it, we define a procedure for automatically deriving a self-monitoring MAS. Such a generated MAS ensures that agents conform to the protocol at run-time, by adding a *monitor* agent that checks that the ongoing conversation is correct w.r.t. the global session type.

The feasibility of the approach has been experimented in Jason for a non-trivial example involving recursive global session types with alternative choice and fork type constructors. Although the main aim of this work is the development of a unit testing framework for MASs, the proposed approach can be also extended to implement a framework supporting self-recovering MASs.

1 Introduction

A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do a meaningful interaction: participants simply cannot communicate effectively.
The development and validation of programs against protocol descriptions could proceed as follows:

- *A programmer specifies a set of protocols to be used in her application.*

...

- *At the execution time, a local monitor can validate messages with respect to given protocols, optionally blocking invalid messages from being delivered.*

This paper starts with a few sentences drawn from the manifesto of Scribble, a language to describe application-level protocols among communicating systems

M. Baldoni et al. (Eds.): DALT 2012, LNAI 7784, pp. 76–95, 2013.

initially designed by Kohei Honda and Gary Brown[1]. The team working on Scribble involves both scientists active in the agent community and scientists active in the session types one. Their work inspired the proposal presented in this paper where multiparty global session types are used on top of the Jason agent oriented programming language for runtime verification of the conformance of a MAS implementation to a given protocol. This allows us to experiment our approach on realistic scenarios where messages may have a complex structure, and their content may change from one interaction to another.

Following Scribble's manifesto, we ensure runtime conformance thanks to a Jason monitor agent that can be automatically generated from the global session type, represented as a Prolog cyclic term. Besides the global session type, the developer must specify the type of the actual messages that are expected to be exchanged during a conversation.

In order to verify that a MAS implementation is compliant with a given protocol, the Jason code of the agents that participate in the protocol is extended seamlessly and automatically. An even more transparent approach would be possible by overriding the underlying agent architecture methods of Jason responsible for sending and receiving messages, which could intercept all messages sent by the monitored agents, and send them to the monitor which could manage them in the most suitable way. In this approach message "sniffing" would have to occur at the Java (API) level, gaining in transparency but perhaps loosing in flexibility.

In this paper we show the feasibility of our approach by testing a MAS against a non-trivial protocol involving recursive global session types with alternative choice and fork type constructors.

The paper is organized in the following way: Section 2 provides a gentle introduction to the global session types we used in our research; Section 3 discusses our implementation of the protocol testing mechanism; Section 4 presents the results of some experiments we have carried out; Section 5 discusses the related literature and outlines the future directions of our work.

2 A Gentle Introduction to Global Session Types for Agents

In this section we informally introduce global session types (global types for short) and show how they can be smoothly integrated in MASs to specify multiparty communication protocols between agents. To this aim, we present a typical protocol that can be found in literature as our main running example used throughout the paper.

Our example protocol involves three different agents playing the roles of a seller `s`, a broker `b`, and a client `c`, respectively. Such a protocol is described by the FIPA AUML interaction diagram [17] depicted in Figure 1: initially, `s` communicates to `b` the intention to sell a certain item to `c`; then the protocol

[1] `http://www.jboss.org/scribble/`

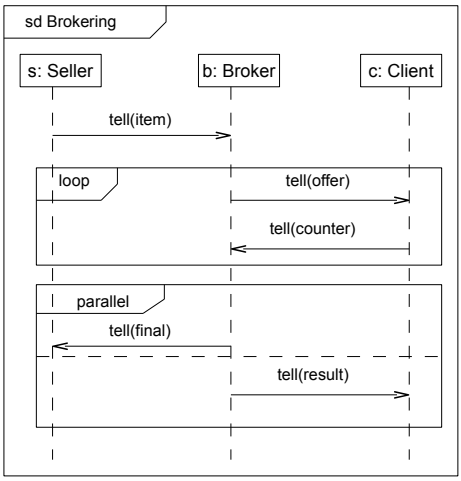

Fig. 1. The Brokering interaction protocol in FIPA AUML

enters a negotiation loop of an arbitrary number n (with $n \geq 0$) of iterations, where `b` sends an offer to `c` and `c` replies with a corresponding counter-offer. After such a loop, `b` concludes the communication by sending in an arbitrary order the message of type `result` to `c`, and of type `final` to `s`.

Even though the AUML diagram of Figure 1 is very intuitive and easy to understand, a more compact and formal specification of the protocol is required to perform verification or testing of a MAS, in order to provide guarantees that the protocol is implemented correctly. Global session types [8,14] have been introduced and studied exactly for this purposes, even though in the more theoretical context of calculi of communicating processes. A global type describes succinctly all sequences of sending actions that may occur during a correct implementation of a protocol.

Depending on the employed type constructors, a global type can be more or less expressive. Throughout this paper we will use a fixed notion of global type, but our proposed approach can be easily adapted for other kinds of global types. The notion of global type we adopt is a slightly less expressive version of that proposed by Deniélou and Yoshida [10] (which, however, allows us to specify the protocol depicted in Figure 1), defined on top of the following type constructors:

– *Sending Actions*: a sending action occurs between two agents, and specifies the sender and the receiver of the message (in our case, the names of the agents, or, more abstractly, the role they play in the communication), and the type of the performative and of the content of the sent message; for instance, `msg(s, b, tell, item)` specifies that agent `s` (the seller) sends the **`tell`** performative to agent `b` (the broker) with content of type `item`.

- *Empty Type*: the constant `end` represents the empty interaction where no sending actions occur.
- *Sequencing*: sequencing is a binary constructor allowing a global type t to be prefixed by a sending action a; that is, all valid sequences of sending actions denoted by `seq(`a`,`t`)` are obtained by prefixing with a all those sequences denoted by t. For instance,

```
seq(msg(alice,bob,tell,ping),
    seq(msg(bob,alice,tell,pong),end))
```

  specifies the simple interaction where first `alice` sends **tell**`(ping)` to `bob`, then `bob` replies to `alice` with **tell**`(pong)`, and finally the interaction stops.
- *Choice*: the choice constructor has variable arity[2] n (with $n \geq 0$) and expresses an alternative between n possible choices. Because its arity is variable we use a list to represent its operands. For instance,

```
choice([
    seq(msg(c,b,tell,counter),end),
    seq(msg(b,s,tell,final),end),
    seq(msg(b,c,tell,result),end)
])
```

  specifies an interaction where either `c` sends **tell**`(counter)` to `b`, or `b` sends **tell**`(final)` to `s`, or `b` sends **tell**`(result)` to `c`.
- *Fork*: the fork binary[3] constructor specifies two interactions that can be interleaved. For instance,

```
fork(
    seq(msg(b,s,tell,final),end),
    seq(msg(b,c,tell,result),end)
)
```

  specifies the interaction where first `b` sends **tell**`(final)` to `s`, and then `b` sends **tell**`(result)` to `c`, or the other way round.

Recursive types: the example types shown so far do not specify any interaction loop, as occurs in the protocol of Figure 1. To specify loops we need to consider recursive global types; for instance, the protocol consisting of infinite sending actions where first `alice` sends **tell**`(ping)` to `bob`, and then `bob` replies **tell**`(pong)` to `alice`, can be represented by the recursive type `T` s.t.

```
T = seq(msg(alice,bob,tell,ping),
        seq(msg(bob,alice,tell,pong),T))
```

If we interpret the equation above syntactically (that is, as a unification problem), then the unique solution is an infinite term (or, more abstractly, an infinite tree) which is *regular*, that is, whose set of subterms is finite. In practice, the unification problem above is solvable in most modern implementations of Prolog,

[2] Arity 0 and 1 are not necessary, but make the definition of predicate `next` simpler.
[3] For simplicity, the operator has a fixed arity, but it could be generalized to the case of n arguments (with $n \geq 2$) as happens for the `choice` constructor.

where cyclic terms are supported; this happens also for the Jason implementation, where Prolog-like rules can be used to derive beliefs that hold in the current belief base[4]. As another example, let us consider the type `T2` s.t.

```
T2 = seq(msg(alice,bob,tell,ping),
         seq(msg(bob,alice,tell,pong),choice([T2,end])))
```

Such a type contains the infinite interaction denoted by `T` above, but also all finite sequences of length $2n$ (with $n \geq 1$) of alternating sending actions `msg(alice,bob,tell,ping)` and `msg(bob,alice,tell,pong)`.

We are now ready to specify the `Brokering` protocol with a global type `BP`, where for sake of clarity we use the auxiliary types `OffOrFork`, `Off`, and `Fork`:

```
BP         = seq(msg(s,b,tell,item),OffOrFork),
OffOrFork  = choice([Off,Fork])
Off        = seq(msg(b,c,tell,offer),
                 seq(msg(c,b,tell,counter),OffOrFork))
Fork       = fork(seq(msg(b,s,tell,final),end),
                  seq(msg(b,c,tell,result),end))
```

Note that for the definition of global types we consider in this paper, the `fork` constructor does not really extend the expressiveness of types: any type using `fork` can be transformed into an equivalent one without `fork`. However, such a transformation may lead to an exponential growth of the type .To see this, let us consider the following type `F`:

```
F          = fork(AliceBob,CarolDave),
AliceBob   = seq(msg(alice,bob,tell,ping),
                 seq(msg(bob,alice,tell,pong),AliceBob))
CarolDave  = seq(msg(carol,dave,tell,ping),
                 seq(msg(dave,carol,tell,pong),CarolDave))
```

Type `F` is equivalent to the following type `AC` that does not contain any `fork`:

```
AC = choice([seq(msg(alice,bob,tell,ping),BC),
             seq(msg(carol,dave,tell,ping),AD)]),
BC = choice([seq(msg(bob,alice,tell,pong),AC),
             seq(msg(carol,dave,tell,ping),BD)]),
AD = choice([seq(msg(alice,bob,tell,ping),BD),
             seq(msg(dave,carol,tell,pong),AD)]),
BD = choice([seq(msg(bob,alice,tell,pong),AD),
             seq(msg(dave,carol,tell,pong),BC)])
```

Formal Definitions

Figure 2 defines the abstract syntax of the global session types that will be used in the rest of the paper. As already explained in the previous section, global types are defined coinductively: GT is the greatest set of regular terms defined by the productions of Figure 2.

[4] Persistency of cyclic terms is supported by the very last version of Jason; since testing of this feature is still ongoing, it has not been publicly released yet.

$$
\begin{aligned}
\text{GT} ::= & \ \text{choice}([\text{GT}_1, \ldots, \text{GT}_n]) \ (n \geq 0) \mid \\
& \ \text{seq}(\text{SA}, \text{GT}) \mid \\
& \ \text{fork}(\text{GT1}, \text{GT1}) \mid \\
& \ \text{end} \\
\text{SA} ::= & \ \text{msg}(\text{AId}_1, \text{AId}_2, \text{PE}, \text{CT})
\end{aligned}
$$

Fig. 2. Syntax of Global Types

The meta-variables AId, PE and CT range over agent identifiers, performatives, and content types, respectively. Content types are constants specifying the types of the contents of messages.

The syntactic definition given so far still contains global types that are not considered useful, and, therefore, are rejected for simplicity. Consider for instance the following type `NC`:

```
NC = choice([NC,NC])
```

Such a type is called *non contractive* (or *non guarded*), since it contains an infinite path with no `seq` type constructors. These kinds of types pose termination problems during dynamic global typechecking. Therefore, in the sequel we will consider only *contractive* global types (and we will drop the term "contractive" for brevity), that is, global types that do not have paths containing only the `choice` and `fork` type constructors. Such a restriction does not limit the expressive power of types, since it can be shown that for every non contractive global type, there exists a contractive one which is equivalent, in the sense that it represents the same set of sending action sequences. For instance, the type `NC` as defined above corresponds to the empty type `end`.

Interpretation of global types. We have already provided an intuition of the meaning of global types. We now define their interpretation, expressed in terms of a `next` predicate, specifying the possible transitions of a global type. Intuitively, a global type represents a state from which several transition steps to other states (that is, other global types) are possible, with a resulting sending action. Consider for instance the type `F` defined by

```
fork(seq(msg(b,s,tell,final),end),
     seq(msg(b,c,tell,result),end))
```

Then there are two possible transition steps: one yields the sending action `msg(b,s,tell,final)` and moves to the state corresponding to the type

```
fork(end,
     seq(msg(b,c,tell,result),end))
```

while the other yields the sending action `msg(b,c,tell,result)` and moves to the state corresponding to the type

```
fork(seq(msg(b,s,tell,final),end),
     end)
```

Predicate `next` is defined below, with the following meaning: if `next(GT1,SA,GT2)` succeeds, then there is a one step transition from the state represented by the global type GT1 to the state represented by the global type GT2, yielding the sending action SA. The predicate is intended to be used with the mode indicators `next(+,+,-)`, that is, the first two arguments are input, whereas the last is an output argument.

```
1 next(seq(msg(S, R, P, CT),GT),msg(S, R, P, C),GT) :-
      has_type(C, CT).
2 next(choice([GT1|_]),SA,GT2) :- next(GT1,SA,GT2).
3 next(choice([_|L]),SA,GT) :- next(choice(L),SA,GT).
4 next(fork(GT1,GT2),SA,fork(GT3,GT2)) :- next(GT1,SA,GT3).
5 next(fork(GT1,GT2),SA,fork(GT1,GT3)) :- next(GT2,SA,GT3).
```

We provide an explanation for each clause:

1. For a sequence `seq(msg(S, R, P, CT),GT)` the only allowed transition step leads to state `GT`, and yields a sending action `msg(S, R, P, C)` where `C` is required to have type `CT`; we assume that all used content types are defined by the predicate `has_type`, whose definition is part of the specification of the protocol, together with the initial global type.
2. The first clause for `choice` states that there exists a transition step from `choice([GT1|_])` to `GT2` yielding the sending action `SA`, whenever there exists a transition step from `GT1` to `GT2` yielding the sending action `SA`.
3. The second clause for `choice` states that there exists a transition step from `choice([_|L])` to `GT` yielding the sending action `SA`, whenever there exists a transition step from `choice(L)` (that is, the initial type where the first choice has been removed) to `GT` yielding the sending action `SA`.
   Note that both clauses for `choice` fail for the empty list, as expected (since no choice can be made).
4. The first clause for `fork` states that there exists a transition from `fork(GT1,GT2)` to `fork(GT3,GT2)` yielding the sending action `SA`, whenever there exists a transition step from `GT1` to `GT3` yielding the sending action `SA`.
5. The second clause for `fork` is symmetric to the first one.

We conclude this section by a claim stating that contractive types ensure termination of the resolution of `next`.

Proposition 1. *Let us assume that* `has_type(`c`,`ct`)` *always terminates for any ground atoms* c *and* ct*. Then,* `next(`gt`,`sa`,`X`)` *always terminates, for any ground terms* gt *and* sa*, and logical variable* X*, if* gt *is a contractive global type.*

Proof. By contradiction, it is straightforward to show that if `next(`gt`,`sa`,`X`)` does not terminate, then gt must contain a (necessarily infinite) path with only `choice` and `fork` constructors, hence, gt is not contractive.

3 A Jason Implementation of a Monitor for Checking Global Session Types

As already explained in the Introduction, the main motivation of our work is a better support for testing the conformance of a MAS to a given protocol, even

though we envisage other interesting future application scenarios (see Section 5). From this point of view our approach can be considered as a first step towards the development of a unit testing framework for MASs where testing, types, and – more generally – formal verification can be reconciled in a synergistic way.

In more detail, given a Jason implementation of a MAS[5], our approach allows automatic generation[6] of an extended MAS from it, that can be run on a set of tests to detect possible deviations of the behavior of a system from a given protocol. To achieve this the developer is required to provide (besides the original MAS, of course) the following additional definitions:

- The Prolog clauses for predicate `next` defining the behavior of the used global types (as shown in Section 2); such clauses depend on the notion of global type needed for specifying the protocol; depending on the complexity of the protocol, one may need to adopt more or less expressive notions of global types, containing different kinds of type constructors, and for each of them the corresponding behavior has to be defined in terms of the `next` predicate. However, we expect the need for changing the definition of `next` to be a rare case; the notion of global type we present here captures a large class of frequently used protocols, and it is always possible to extend the testing unit framework with a collection of predefined notions of global types among which the developer can choose the most suitable one.
- The global type specifying the protocol to be tested; this can be easily defined in terms of a set of unification equations.
- The clauses for the `has_type` predicate (already mentioned in Section 2), defining the types used for checking the content of the messages; also in this case, a set of predefined primitive types could be directly supported by the framework, leaving to the developer the definition of the user-defined types.

The main idea of our approach relies on the definition of a centralized *monitor* agent that verifies that a conversation among any number of participants is compliant with a given global type, and warns the developer if the MAS does not progress. Furthermore, the code of the agents of the original MAS requires minimal changes that, however, can be performed in an automatic way.

In the sequel, we describe the code of the monitor agent, and the changes applied to all other agents (that is, the participants of the implemented protocol).

3.1 Monitor

We illustrate the code for the monitor by using our running brokering example. The monitor can be automatically generated from the global type specification in a trivial way. The global type provided by the developer is simply a conjunction *UnifEq* of unification equations of the form $X = GT$, where X is a logical variable, and GT is a term (possibly containing logical variables) denoting a global type. The use of more logical variables is allowed for defining auxiliary

[5] We assume that the reader is familiar with the AgentSpeak language [20].
[6] Its implementation has not been completed yet.

types that make the definition of the main type more readable. Then from *UnifEq* the following Prolog rule is generated:

`initial_state(`X`) :-` *UnifEq*`.`

where X is the logical variable contained in *UnifEq* corresponding to the main global type. The definition of the type of each message content must be provided as well. In fact, the protocol specification defines also the expected types (such as `item`, `offer`, `counter`, `final` and `result`) for the correct content of all possible messages. For example, the developer may decide that the type `offer` defines all terms of shape `offer(Item, Offer)`, where `Item` is a string and `Offer` is an integer; similarly, the type `item` corresponds to all terms of shape `item(Client, Item)` where both `Client` and `Item` are strings.

Consequently, the developer has to provide the following Prolog rules that formalize the descriptions given above:

```
has_type(offer(Item, Offer), offer) :-
   string(Item) & int(Offer).
has_type(item(Client, Item), item) :-
   string(Client) & string(Item).
```

The monitor keeps track of the runtime evolution of the protocol by saving its current state (corresponding to a global type), and checking that each message that a participant would like to send, is allowed by the current state. If so, the monitor allows the participant to send the message by explicitly sending an acknowledgment to it. We explain how participants inform the monitor of their intention to send a message in Section 3.2.

The correctness of a sending action is directly checked by the `next` predicate, that also specifies the next state in case the transition is correct. In other words, verifying the correctness of the message sent by `S` to `R` with performative `P` and content `C` amounts to checking if it is possible to reach a `NewState` from the `CurrentState`, yielding a sending action `msg(S, R, P, C)` (`type_check` predicate).

```
/* Monitor's initial beliefs and rules */

// user-defined predicates
initial_state(Glob) :-
   Merge = choice([Off,Fork]) &
   Off= seq(msg(b, c, tell, offer),
            seq(msg(c, b, tell, counter), Merge)) &
   Fork= fork(seq(msg(b, s, tell, final),end),
              seq(msg(b, c, tell, result),end)) &
   Glob = seq(msg(s, b, tell, item),Merge).

has_type(offer(Item, Offer), offer) :-
   string(Item) & int(Offer).
has_type(counter(Item, Offer), counter) :-
   string(Item) & int(Offer).
has_type(final(Res, Client, Item, Offer), final) :-
   string(Res) & string(Client) & string(Item) & int(Offer).
has_type(result(Res, Item, Offer), result) :-
   string(Res) & string(Item) & int(Offer).
has_type(item(Client, Item), item) :-
   string(Client) & string(Item).
// end of user-defined predicates

timeout(4000).
```

```
type_check(msg(S, R, P, C), NewState) :-
   current_state(CurrentState) &
   next(CurrentState, msg(S, R, P, C), NewState).

// Rules defining the next predicate follow
........
```

The monitor prints every information relevant for testing on the console with the `.print` internal action. The `.send(R, P, C)` internal action implements the asynchronous delivery of a message with performative `P` and content `C` to agent `R`.

A brief description of the main plans follow.

- Plan `test` is triggered by the initial goal `!test` that starts the testing, by setting the current state to the initial state.
- Plan `move2state` upgrades the belief about the current state.
- Plan `successfulMove` is triggered by the `!type_check_message(msg(S, R, P, C))` internal goal. If the `type_check(msg(S, R, P, C), NewState)` context is satisfied, then `S` is allowed to send the message with performative `P` and content `C` to `R`. The state of the protocol changes, and `monitor` notifies `S` that the message can be sent.
- Plan `failingMoveAndProtocol` is triggered, like `successfulMove`, by the `!type_check_message(msg(S, R, P, C))` internal goal. It is used when `successfulMove` cannot be applied because its context is not verified. This means that `S` is not allowed to send message `P` with content `C` to `R`, because a dynamic type error has been detected: the message does not comply with the protocol.
- Plan `messageReceptionOK` is triggered by the reception of a tell message with `msg(S, R, P, C)` content; the message is checked against the protocol, and the progress check is activated (`!check_progress` succeeds either if a message is received before a default timeout, or if the timeout elapses, in which case `!check_progress` is activated again: `.wait(+msg(S1, R1, P1, C1), MS, Delay)` suspends the intention until `msg(S1, R1, P1, C1)` is received or `MS` milliseconds have passed, whatever happens first; `Delay` is unified to the elapsed time from the start of `.wait` until the event or timeout).

All plans whose context involves checking the current state and/or whose body involves changing it are defined as atomic ones, to avoid problems due to interleaved check-modify actions.

```
/* Initial goals */

!test.

/* Monitor's plans */

@test[atomic]
+!test : initial_state(InitialState)
   <- +current_state(InitialState).

@move2state[atomic]
```

```
+!move_to_state(NewState) : current_state(LastState)
   <- -current_state(LastState);
      +current_state(NewState).

@successfulMove[atomic]
+!type_check_message(msg(S, R, P, C)) : type_check(msg(S, R, P, C), NewState)
   <- !move_to_state(NewState);
      .print("\nMessage ", msg(S, R, P, C), "\nleads to state ", NewState, "\n");
      .send(S, tell, ok_check(msg(S, R, P, C))).

@failingMoveAndProtocol
+!type_check_message(msg(S, R, P, C)) : current_state(Current)
   <- .print("\n*** DYNAMIC TYPE-CHECKING ERROR ***\nMessage ", msg(S, R, P, C),
      "\ncannot be accepted in the current state ", Current, "\n");
       !move_to_state(failure).

@messageReceptionOK
+msg(S, R, P, C)[source(S)]: true
   <- -msg(S, R, P, C)[source(S)];
      !type_check_message(msg(S, R, P, C));
      !check_progress.

+!check_progress : timeout(MS)
   <- .wait({+msg(S1, R1, P1, C1)}, MS, Delay);
      !aux_check_progress(Delay).

+!aux_check_progress(Delay) : timeout(MS) & Delay < MS.

+!aux_check_progress(Delay) : timeout(MS) & current_state(Current) & Delay >= MS
   <- .print("\n*** WARNING ***\nNo progress for ", Delay, " milliseconds
      in the current state ", Current, "\n");
      !check_progress.
```

3.2 Participants

We assume that participants interact via asynchronous exchange of messages with `tell` performatives.

To keep the implementation as general and flexible as possible, in the participants' code extended as explained below we use the `Perf` logical variable where the message performative is expected. Under the assumption that only `tell` performatives will be used, `Perf` will always be bound to the `tell` ground atom.

Only two changes are required to the code of participants:

1. `.send` is replaced by `!my_send` and
2. two plans are added for managing the interaction with the monitor.

The first plan is triggered by the `!my_send` internal goal; `my_send` has the same signature as the `.send` internal action, but, instead of sending a message with performative `Perf` and `Content` to `Receiver`, it sends a `tell` message to the monitor in the format `msg(Sender, Receiver, Perf, Content)`. When received, this message will be checked by the monitor against the global type, as explained in Section 3.1.

The second plan is triggered by the reception of the monitor's message that allows the agent to actually send `Content` to `Receiver`, by means of a message with performative `Perf`. In reaction to the reception of such a message, the agent sends the corresponding message to the expected agent.

```
/* Plans for runtime type checking */

+!my_send(Receiver, Perf, Content) : true
   <- .my_name(Sender);
      .send(monitor, tell, msg(Sender, Receiver, Perf, Content)).

+ok_check(msg(Sender, Receiver, Perf, Content))[source(monitor)] : true
   <- -ok_check(msg(Sender, Receiver, Perf, Content))[source(monitor)];
      .send(Receiver, Perf, Content).
```

3.3 Discussion

Alternative implementations. We opted to implement the proof-of-concept of our approach by extending the code of the existing participants rather than modifying the code of the Jason interpreter, because this was the simplest and quickest solution we could devise for developing a prototype, and easily experimenting different design choices. However, the same results could be obtained by directly modifying the `.send` internal action by overriding the underlying agent architecture methods of Jason responsible for sending and receiving messages.

This solution would not require any modification of the code of the participants, and would allow the monitor to forward the message, when correct, directly to the recipient agent, thus reducing the number of interactions required among agents.

Another interesting solution would consist in creating a monitor agent for each agent participating to the interaction, thus avoiding the communication problems of the centralized approach where the unique monitor is required to exchange a large amount of messages with the other agents; however, this solution requires to project the global session type to end-point types (a.k.a. local types), specifying the expected behavior of each single agent involved in the interaction. Depending on the considered notion of global type, it might be non trivial to find an efficient and complete projection algorithm.

Global type transition. We have already shown that the `next` predicate is ensured to terminate on contractive global types; however, a developer may erroneously define a non contractive type for testing its system. Fortunately, there exist algorithms for automatically translating a non contractive global type into an equivalent contractive one.

Another issue concerns non deterministic global types, that is, global types where transitions are not deterministic. Consider for instance the following global type:

```
fork(seq(msg(alice,bob,tell,ping),
         seq(msg(bob,alice,tell,pong),end)),
     seq(msg(alice,bob,tell,ping),
         seq(msg(alice,bob,tell,bye),end)))
```

In this case the `next` predicate has to guess which of the two operand types must progress upon reception of the message matching with `msg(alice,bob,tell,ping)`; this means that in case of non deterministic global types the monitor may detect false positives. To avoid this problem one could determinize the type, but

depending on the considered notion of global type, it would not be easy, or even possible, to devise a determinization algorithm. Alternatively, the monitor could store the whole sequence of received sending actions to allow backtracking in case of failure, thus making the testing procedure much less efficient.

Finally, it is worth mentioning that the proposed approach makes an efficient use of memory space if the initial global type does not contain loops with the `fork` constructor. In this case the space required by a global type representing an intermediate state is bounded by the size of the initial global type; since only one type at a time is kept in the belief base of the monitor, this implies a significant space optimization when the total number of all possible states is exponential w.r.t. the size of the initial global type. As already pointed out, this consideration does not apply to types with loops involving the `fork` constructor, like in the following example:

```
T = fork(seq(msg(alice,bob,tell,ping),T),
         seq(msg(bob,alice,tell,pong),T)).
```

In this case the term grows at each transition step (and there are cases where the type cannot be simplified to a smaller one); however, we were not able to come up with examples of realistic protocols that require types with `fork` in a loop to be specified.

4 The Framework at Work

In this section we show the actual functioning of our framework by discussing the experiments we made with the brokering global type. We show the correct code of seller (s), broker (b) and client (c) apart from the fragments common to all of them and discussed in Section 3, and omitting the definition of intuitive predicates, and then we discuss how the framework works with both correct and buggy code.

Seller. The seller starts the conversation (it has a `!start` initial goal) by sending a message to the broker telling that it wants to sell `orange` to `c`. It has a plan triggered by the reception of the final result of the negotiation, whose body is empty, and no initial beliefs.

```
/* Plans */

+!start : true
   <- !my_send(b, tell, item(c, orange)).

+final(Res, Client, Item, Offer)[source(Broker)] : true.
```

Broker. The broker has no initial goals and its policy is the following:

- whatever the item to trade, and the client with whom trading, it proposes to sell it at an initial price stored in its belief base (10 euros for a crate of oranges when trading with `c`).
- Depending on the counter offer it receives, three situations may take place:

1. The counter offer is in a range that leaves room for negotiation. The broker makes an offer with price decremented by one with respect to the previous offered one (first plan triggered by `+counter(Item, Offer)`).
2. The counter offer is too low and there is no room for negotiating. The final decision (`noDeal`) is sent both to the seller and to the client (second plan triggered by `+counter(Item, Offer)`).
3. The counter offer can be accepted. The final decision (`ok`) is sent both to the seller and to the client. We do not show the plan for this case, since it is very similar to the previous one.

```
/* Initial beliefs and rules */

initial_offer(c, orange, 11).
acceptable_offer(c, orange, 6).

/* Plans */

+item(Client, Item)[source(s)] : initial_offer(Client, Item, Offer)
   <- +current_offer(Client, Item, Offer);
      !my_send(Client, tell, offer(Item, Offer)).

+counter(Item, Offer)[source(Client)] : acceptable_offer(Client, Item, Min)
                                        & Offer < Min & Offer > Min-4
   <- !decrement(Client, Item, NewOffer);
      !my_send(Client, tell, offer(Item, NewOffer)).

+counter(Item, Offer)[source(Client)] : acceptable_offer(Client, Item, Min)
                                        & Offer <= Min-4
   <- !my_send(Client, tell, result(noDeal, Item, Offer));
      !my_send(s, tell, final(noDeal, Client, Item, Offer)).
```

Client. The client has a reactive behavior: whatever the offer it receives, the client answers with a counter offer depending on the `initial_counter_offer` belief in its belief base, and increments it by one at any interaction step, until it receives the result of the negotiation.

```
/* Initial beliefs and rules */

initial_counter_offer(b, orange, 3).

/* Plans */

+offer(Item, Offer)[source(Broker)] : initial_counter_offer(Broker, Item, Initial)
   <-  -initial_counter_offer(Broker, Item, Initial);
       -offer(Item, Offer)[source(Broker)];
       +current_counter_offer(Broker, Item, Initial);
       !my_send(Broker, tell, counter(Item, Initial)).

+offer(Item, Offer)[source(Broker)] : true
   <- -offer(Item, Offer)[source(Broker)];
      !increment(Broker, Item, NewOffer);
      !my_send(Broker, tell, counter(Item, NewOffer)).

+result(Res, Item, Offer)[source(Broker)]: true.
```

4.1 Running the Example

When running the MAS consisting of agents `monitor`, `s`, `b`, and `c`, we obtain console messages like those shown below (we only show the first operators of the

printed states, for space constraints; we use "_" for the dropped text, since "..." is part of the cyclic term representation; `Msg -> St` means that the agent moves to state `St` upon reception of `Msg`). The conversation complies with the global type and a state that is equivalent to `end` is reached. Since we do not model the notion of protocol termination, the monitor cannot know that the protocol terminated successfully, and keeps watching the conversation and issues warning messages every `M` seconds. The developer can easily verify that no messages are sent because no more messages had to be sent in state `fork(end,end)`.

```
[monitor]
msg(s,b,tell,item(c,orange)) -> choice([...seq(msg(b,c,tell,offer),_)
....

[monitor]
msg(b,c,tell,offer(orange,9)) -> seq(msg(c,b,tell,counter),choice([_]))

[monitor]
msg(c,b,tell,counter(orange,5)) -> choice([...seq(msg(b,c,tell,offer),_)])

[monitor]
msg(b,c,tell,offer(orange,8)) -> seq(msg(c,b,tell,counter),choice([_]))

[monitor]
msg(c,b,tell,counter(orange,6)) -> choice([...seq(msg(b,c,tell,offer),_)])

[monitor]
msg(b,c,tell,result(ok,orange,6)) -> fork(seq(msg(b,s,tell,final),end),end)

[monitor]
msg(b,s,tell,final(ok,c,orange,6)) -> fork(end,end)

[monitor]
*** WARNING ***
No progress for 4001 milliseconds in the current state fork(end,end)
```

Bug 1. Let us suppose that the second plan for dealing with offers in the client's code, is the following:

```
+offer(Item, Offer)[source(Broker)] : true
    <- -offer(Item, Offer)[source(Broker)];
       !increase(Broker, Item, NewOffer);
       !my_send(Broker, tell, offer(Item, NewOffer));
       !my_send(Broker, tell, anotherOffer(Item, NewOffer)).
```

Instead of sending a counter offer, the client sends an `offer` followed by a message with unknown type. The console messages we obtain in this case are shown below.

```
...

[monitor]
msg(b,c,tell,offer(orange,8)) -> seq(msg(c,b,tell,counter),choice([_]))

[monitor]
*** DYNAMIC TYPE-CHECKING ERROR ***
msg(c,b,tell,offer(orange,4)) cannot be accepted in
seq(msg(c,b,tell,counter), choice([_]))

[monitor]
*** DYNAMIC TYPE-CHECKING ERROR ***
msg(c,b,tell,anotherOffer(orange,4)) received when no ongoing protocol
```

The monitor notifies two dynamic type checking errors: the first one due to the unexpected `offer` message, and the second one due to the message received after the protocol testing failed. The message that caused the failure and the current global type state are shown. When a protocol fails, warnings about lack of progress are suppressed.

The developer can either fix the code of the agent that sent the message or the specification of the global type, depending on where the error was.

Bug 2. The client has a `!start` initial goal, hence it autonomously starts to interact with the broker before the previous messages that the protocol enforces have been sent:

```
/* Plans */

+!start : initial_counter_offer(Broker, Item, Initial)
   <-  -initial_counter_offer(Broker, Item, Initial);
       +current_counter_offer(Broker, Item, Initial);
       !my_send(Broker, tell, counter(Item, Initial)).
```

The monitor prints out the following message:

```
[monitor]
*** DYNAMIC TYPE-CHECKING ERROR ***
msg(c,b,tell,counter(orange,3)) cannot be accepted in
seq(msg(s,b,tell,item),choice([_]))
```

Bug 3. We deleted all the plans triggered by the reception of `+counter(Item, Offer)[source(Client)]` from the broker's code, making the broker agent unable to react to a counter offer. The state of the protocol printed out by the monitor in its warning message helps the MAS developer in identifying the agent that is expected to send a message at that point of the conversation.

```
[monitor]
msg(s,b,tell,item(c,orange)) -> state choice([...seq(msg(b,c,tell,offer),_])

[monitor]
msg(b,c,tell,offer(orange,11)) -> state seq(msg(c,b,tell,counter),choice([_]))

[monitor]
msg(c,b,tell,counter(orange,3)) -> state choice([...seq(msg(b,c,tell,offer),_])

[monitor]
*** WARNING ***
No progress for 4000 ms in choice([...seq(msg(b,c,tell,offer),_)])
```

We run the MAS with different values for the broker's initial and acceptable offers, and with various communication errors besides those described in the paragraphs above, always obtaining the expected result.

5 Related and Future Work

Our work represents a first step in two directions: extending an existing agent programming language with session types, and supporting testing of protocol conformance within a MAS. In this section we consider the related works in both areas, discuss the (lack of) proposals of integrating session types in existing MASs frameworks, and outline possible extensions of our work.

Session types on top of existing programming languages. The integration of session types into existing languages is a recent activity, dating back to less than ten years ago for object oriented calculi, and less than five years for declarative ones. The research field is very lively and open, with the newest proposals published just a few months ago.

Session types have been integrated into object calculi starting from 2005 [11,12]. The first full implementation of a language and run-time for session-based distributed programming on top of Java, featuring asynchronous message passing, delegation, session subtyping and interleaving, combined with class downloading and failure handling, dates back to 2008 [16]. More recently, a Java language extension has been proposed, that counters the problems of traditional event-based programming with abstractions and safety guarantees based on session types [15].

Closer to our work on declarative languages, the paper [21] discusses how session types have been incorporated into Haskell as a standard library that allows the developer to statically verify the use of the communication primitives provided without an additional type checker, preprocessor or modification to the compiler. A session typing system for a featherweight Erlang calculus that encompasses the main communication abilities of the language is presented in [19]. Structured types are used to govern the interaction of Erlang processes, ensuring that their behavior is safe with respect to a defined protocol.

Protocol representation and verification in MASs. Because of the very nature of MASs as complex systems consisting of autonomous communicating entities that must adhere to a given protocol in order to allow the MAS correct functioning, the problem of how representing interaction protocols has been addressed since the dawning of research on MASs (one of the most well known outcomes being FIPA AUML interaction diagrams [17]), and the literature on protocol conformance verification is extremely rich.

Although a bit dated, [6] still represents one of the most valuable contributions to *verification of a priori conformance*. In that paper the authors propose an approach based on the theory of formal languages to formally prove the interoperability of two policies (the actual protocol implementations), each of which is compliant with a protocol specification.

The problem of *verifying the compliance of protocols at run time* has been tackled – among others – within the SOCS project[7], where the SCIFF computational logic framework [1] is used to provide the semantics of social integrity constraints. Such a semantics is based on abduction: expectations on the possibly observable, yet unknown, events are modeled as abducibles and social integrity constraints are represented as integrity constraints. To model MAS interaction, expectation-based semantics specifies the links between the observed events and the expected ones. The recent paper "Modelling Interactions via Commitments and Expectations" [23] discusses that and related approaches. Although aimed at testing run-time conformance of an actual conversation with respect to a given protocol, our approach differs from the expectation-based one in many respects,

[7] `http://lia.deis.unibo.it/research/projects/SOCS/`

including the lack of notion of expectation in the agent language, and the implementation of the testing mechanism in a seamless way on top of an existing and widespread agent-oriented programming language. As far as formalisms for representing agent interaction protocols are concerned, the reader may find a concise but very good survey in Section 4 of [22] where the authors propose a commitment-based semantics of protocols.

Our approach is currently limited to the runtime verification of the MAS compliance to the interaction protocol, but the exploitation of session types as the formalism to represent protocols allows us to take advantage of all the results achieved in the session types research field, which include session subtyping and algorithms for static verification of protocol properties such as safety and liveness. The ability to specify the type of messages (`has_type(`c`,`ct`)` predicate) in order to relate actual messages to messages specified in the protocol, usually given at a more abstract level, is a characterizing feature of our approach and seems to be supported by none of the proposals mentioned above.

Session Types and MASs. As demonstrated for example by the Scribble language mentioned in the Introduction and by [13], using session types to represent and verify protocol conformance inside MASs is not a new idea but, to the best of our knowledge, no attempts of taking advantage of global session types to verify MASs programmed in some widespread agent oriented programming languages had been made so far, and our proposal is an original one.

Future extensions. Some extensions to our work have already been implemented in the last few months: in [2] we explored the theoretical foundations of our framework and we introduced a concatenation operator that allows a significant enhancement of the expressive power of our global types. In [3] we further empowered our formalism with a mechanism for easily expressing constrained shuffle of message sequences like the alternating bit protocol discussed in [10]; accordingly, we modified the semantics of the new introduced feature, and showed the expressive power of these "constrained global types". With respect to this extension, we are currently exploring the work of Baier, et al. on Constraint Automata [4,5] that offers a transition system using synchronization constraints and data constraints to specify behavior and concurrent protocols as automata models. Constraint Automata are compositional, i.e., more complex protocols/behaviors can be constructed as a composition of simpler protocols/behaviors, which is a common goal with our work.

Our work can be further extended in many ways. Besides the specific issues mentioned in Section 3, and the fully automatic generation of the monitor and participants code, our short term goals include analyzing how our approach could be extended to other Prolog-based agent-programming languages, such as GOAL [7] or 2APL [9], and designing more complex protocols to stress-test our system and provide a quantitative assessment of its runtime behavior and scalability.

In the medium term, we plan to work for evolving our mechanism towards a framework supporting self-recovering MASs. This evolution would require to modify the way we extend the code of the participant agents, in order to automatically select other messages to send in the current state, if any, in case

the monitor realizes that the chosen one does not respect the protocol. Default recovery actions for the situation where no other choices are available, should be defined as well. In such a context – more oriented towards verification of interoperability of deployed systems rather than testing of systems-to-be –, agents might advertise to the monitor the services they offer and the protocols to follow in order to obtain them. Besides ensuring the protocol's compliance, the monitor could then act as a repository of <*service specification, protocol specification*> couples, helping agents to locate services in an open MAS in a similar way the Universal Description, Discovery and Integration (UDDI) registry does for web services.

In the long term, the integration of ontology-based meaning into protocol specifications, leading to "ontology-aware session types", will be addressed. Our previous work on CooL-AgentSpeak [18] will represent the starting point for that extension.

Acknowledgments. We are grateful to J. F. Hübner and R. H. Bordini for their effort in making cyclic terms in Jason belief base persistent, thus making the implementation of our monitor agent possible. We also thank the anonymous reviewers for their careful reading of the paper and for the valuable suggestions provided to improve its quality.

References

1. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: The *S*CIFF Abductive Proof-Procedure. In: Bandini, S., Manzoni, S. (eds.) AI*IA 2005. LNCS (LNAI), vol. 3673, pp. 135–147. Springer, Heidelberg (2005)
2. Ancona, D., Barbieri, M., Mascardi, V.: Global Types for Dynamic Checking of Protocol Conformance of Multi-Agent Systems (Extended Abstract). In: Massazza, P. (ed.) ICTCS 2012, pp. 39–43 (2012)
3. Ancona, D., Barbieri, M., Mascardi, V.: Constrained global types for dynamic checking of protocol conformance in multi-agent systems. In: SAC 2013. ACM (to appear, 2013)
4. Arbab, F., Baier, C., de Boer, F.S., Rutten, J.J.M.M.: Models and temporal logical specifications for timed component connectors. Software and System Modeling 6(1), 59–82 (2007)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. 61(2), 75–113 (2006)
6. Baldoni, M., Baroglio, C., Martelli, A., Patti, V.: Verification of Protocol Conformance and Agent Interoperability. In: Toni, F., Torroni, P. (eds.) CLIMA VI. LNCS (LNAI), vol. 3900, pp. 265–283. Springer, Heidelberg (2006)
7. Braubach, L., Pokahr, A., Moldt, D., Lamersdorf, W.: Goal Representation for BDI Agent Systems. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2004. LNCS (LNAI), vol. 3346, pp. 44–65. Springer, Heidelberg (2005)
8. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)

9. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems 16(3), 214–248 (2008)
10. Deniélou, P.-M., Yoshida, N.: Multiparty Session Types Meet Communicating Automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012)
11. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)
12. Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S.: A Distributed Object-Oriented Language with Session Types. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 299–318. Springer, Heidelberg (2005)
13. Grigore, C., Collier, R.: Supporting agent systems in the programming language. In: WI/IAT, pp. 9–12. IEEE Computer Society (2011)
14. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL 2008, pp. 273–284. ACM (2008)
15. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-Safe Eventful Sessions in Java. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 329–353. Springer, Heidelberg (2010)
16. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 516–541. Springer, Heidelberg (2008)
17. Huget, M.-P., Bauer, B., Odell, J., Levy, R., Turci, P., Cervenka, R., Zhu, H.: FIPA modeling: Interaction diagrams. Working Draft Version (July 02, 2003), http://www.auml.org/auml/documents/ID-03-07-02.pdf
18. Mascardi, V., Ancona, D., Bordini, R.H., Ricci, A.: CooL-AgentSpeak: Enhancing AgentSpeak-DL agents with plan exchange and ontology services. In: IAT 2011, pp. 109–116. IEEE Computer Society (2011)
19. Mostrous, D., Vasconcelos, V.T.: Session Typing for a Featherweight Erlang. In: De Meuter, W., Roman, G.-C. (eds.) COORDINATION 2011. LNCS, vol. 6721, pp. 95–109. Springer, Heidelberg (2011)
20. Rao, A.S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
21. Sackman, M., Eisenbach, S.: Session types in Haskell: Updating message passing for the 21st century. Technical report, Imperial College, Department of Computing (2008), http://spiral.imperial.ac.uk:8080/handle/10044/1/5918
22. Singh, M.P., Chopra, A.K.: Correctness Properties for Multiagent Systems. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 192–207. Springer, Heidelberg (2010)
23. Torroni, P., Yolum, P., Singh, M.P., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P.: Modelling interactions via commitments and expectations. In: Handbook of Research on Multi-Agent Systems: Semantics and Dynamics of Organizational Models. IGI Global (2009)